1 Introduction

Suppose we want to provide a storage service to a large group of users. Obviously we want to make sure that the data is safe, so we need to account for the fact that some storage nodes might fail. We also want to do that with minimal storage overhead.

Coding theory presents multiple schemes that can solve this situation well. Things become more complicated if users want to modify their data. Suppose users want to perform some insertions and deletions in their data. With many coding schemes, updating the data would be difficult, however this paper introduces schemes that can augment any linear coding scheme to deal with such a situation.

1.1 Coding overview

The main idea of coding is to transform an information vector so that its new form would have the desired attributes. In coding the data vector consists of elements from a finite field. A coding scheme has an encoding and decoding function defined like this:

Definition 1.1. In coding scheme an encoding function is any function of the form:

\[ E : \mathbb{F}_{q_1}^\ell \rightarrow \mathbb{F}_{q_2}^\ell \]

where \( \mathbb{F}_q^\ell \) is a set of all vector of size \( \ell \) over a finite field of size \( q \). Decoding function is just the inverse of \( E \).

If we desire data security (from corruption during storage or transmission) then generally the encoding function is defined in such way that the output vector is larger than the input vector and each possible output vector has a certain "neighborhood" such that the probability of the original coded vector being corrupted beyond its associated neighborhood is minimized. Let’s look at an example next.
Suppose we are trying to store data in a server and we perform regular data validation. Additionally suppose that between validations at most one bit might get corrupted and have its value changed. Here is a possible encoding function that would allow us to deal with such situation:

**Example 1.1.** An example of a simple binary coding function $E: \mathbb{F}_2^2 \rightarrow \mathbb{F}_2^5$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$E(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000</td>
</tr>
<tr>
<td>01</td>
<td>01110</td>
</tr>
<tr>
<td>10</td>
<td>10111</td>
</tr>
<tr>
<td>11</td>
<td>11001</td>
</tr>
</tbody>
</table>

In this example the Hamming distance (the number of positions in which the strings differ) between any two coded binary strings is at least 3. When a single bit gets changed the hamming distance of the new string will be at most 1 away from the original string, so we can easily identify what the original string was. In this case the neighborhood of every output string consists of all the strings with hamming distance at most 1 to that string. The neighborhoods look like this:

**Figure 1.** The neighborhoods of the possible output strings of the aforementioned encoding function.

A linear code is just any coding scheme with the additional property: $E(x + y) = E(x) + E(y)$ for any $x$ and $y$. There are various commonly used linear coding schemes for different purposes and this paper presents an augmentation that would allow any of them to account for insertions and deletions.
1.2 Problem statement

Suppose we have $B$ users and $n$ nodes and they can all communicate with each other. Each user has a data vector with elements taking values from a finite field of size $q$, denoted as $\mathbb{F}_q$. We also have an encoding function $\text{ENCODE} : \mathbb{F}_q^B \rightarrow \mathbb{F}_q^n$. If $n$ is too small for our desired encoding function, then we can just pick some $\alpha$ and define the encoding function as: $\text{ENCODE} : \mathbb{F}_q^B \rightarrow \mathbb{F}_q^{n \times \alpha}$, but that case is easy to deal with and would just complicate this report, so we are not going to analyze it.

Obviously we can extend this function to take $\ell$ bits from each user and convert them to $\ell$ vectors for each of the $n$ nodes. The data of users will be denoted as a tuple of vectors: $\left( x^{(1)}, x^{(2)}, \ldots, x^{(B)} \right)$. The data stored in nodes will be:

$$\text{ENCODE}(\ell) \left( x^{(1)}, x^{(2)}, \ldots, x^{(B)} \right) = \left( \text{ENCODE}(x^{(1)}_1, x^{(2)}_1, \ldots, x^{(B)}_1), \right. \left. \text{ENCODE}(x^{(1)}_2, x^{(2)}_2, \ldots, x^{(B)}_2), \ldots, \right. \left. \text{ENCODE}(x^{(1)}_{\ell}, x^{(2)}_{\ell}, \ldots, x^{(B)}_{\ell}) \right)$$

**Example 1.2.** A simple illustrative linear encoding scheme. Here we have 2 users and 3 nodes. Each data unit will take a value from $\mathbb{F}_5$. The encoding function is:

$$\text{ENCODE}(\ell) \left( x^{(1)}, x^{(2)} \right) = \left( x^{(1)}, x^{(2)}, x^{(1)} + x^{(2)} \right)$$

And the data of the users and nodes will be:

<table>
<thead>
<tr>
<th>user 1</th>
<th>user 2</th>
<th>ENCODE(\ell)(x^{(1)}, x^{(2)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 4, 2, 0, 2)</td>
<td>(3, 2, 1, 3, 4)</td>
<td>(1, 4, 2, 0, 2)</td>
</tr>
<tr>
<td>node 1</td>
<td>node 2</td>
<td>node 3</td>
</tr>
<tr>
<td>(1, 4, 2, 0, 2)</td>
<td>(3, 2, 1, 3, 4)</td>
<td>(4, 1, 3, 3, 1)</td>
</tr>
</tbody>
</table>

Now suppose users perform deletions and insertions on their data. Each node will need to be able to update its data, so each node will need to know at least what operation was performed. However the nodes don’t store the data of users, but instead store some function of the user data. For many functions it might not be possible for the node to update its data without acquiring the entire current state of the user who performed the operation. In fact it can be proven for some linear coding schemes (for example the simple scheme in the example above) that the amount of bits that needs to be communicated in the worst case will be lower-bounded by $\Omega(\ell \log q)$. This paper presents some augmentations that would permit asymptotically faster update with any linear coding scheme.
2 Intermediate Coding Schemes

The issue with insertions and deletions is that the way the encoded data changes depends not only on the location of deletion/insertion and the value of element deleted/inserted, but also on the elements that follow that location. The following schemes circumvent this issue by transforming the user data, before feeding it into the encoding function. For that purpose, each user $s$ keeps track of his $\ell \times \ell$ transformation matrix $A^{(s)}$ and the data we store in the nodes will be: $\text{ENCODE}(\ell)(x^{(1)}A^{(1)}, x^{(2)}A^{(2)}, \ldots, x^{(B)}A^{(B)})$. The desired effect can be achieved with various different matrices like Vandermonde, Cauchy and permutation matrices, however we are going to look at only permutation matrices since they are by far the simplest and provide the most obvious benefit.

2.1 Scheme based on permutation matrices

The following scheme achieves our goal by preventing the reordering of data caused by insertions and deletions. When an element is deleted, we change its value to 0, and when an element is inserted, we place it in some free "slot", preferably the one where its previous element was deleted the latest. In order for the user to be able to reconstruct the data, he needs to keep track of the operations, so each operation also incurs additional storage overhead.

Suppose user $s$ wants to perform some operations. Initially his transformation matrix $A^{(s)}$ is an identity matrix. Suppose he performs a deletion at position $i$. In this case his own data gets modified as expected, however in $A^{(s)}$ the $i$-th row should be shifted to the bottom. As a result $x^{(s)}A^{(s)}$ will correspond to the above description. On the other hand in the case of insertion, we move the last row to the $i$-th position.

Let’s denote the user data after insertion/deletion as: $\hat{x}^{(s)}$ and the user’s new transformation matrix as $\hat{A}^{(s)}$. After the operation the data in the nodes will be given by:

$$\text{ENCODE}(\ell)(x^{(1)}A^{(1)}, x^{(2)}A^{(2)}, \ldots, \hat{x}^{(s)}\hat{A}^{(s)}, \ldots, x^{(B)}A^{(B)})$$

Thanks to the linearity of the encoding function, it is easy to calculate the difference between the current encoded data and the data we are supposed to update to. It will be:

$$\text{ENCODE}(\ell)(x^{(1)}A^{(1)}, x^{(2)}A^{(2)}, \ldots, \hat{x}^{(s)}\hat{A}^{(s)}, \ldots, x^{(B)}A^{(B)}) - \text{ENCODE}(\ell)(x^{(1)}A^{(1)}, x^{(2)}A^{(2)}, \ldots, x^{(s)}A^{(s)}, \ldots, x^{(B)}A^{(B)}) = \text{ENCODE}(\ell)(0, 0, \ldots, \hat{x}^{(s)}\hat{A}^{(s)} - x^{(s)}A^{(s)}, \ldots, 0)$$

This is where the transformation matrices help us out. After either insertion or deletion the change in transformed data will look like this:

$$\hat{x}^{(s)}\hat{A}^{(s)} - x^{(s)}A^{(s)} = (0, 0, \ldots, d, \ldots, 0)$$
Note how there is only one non-zero element. We will clarify the details below with an example. For the nodes to be able to update their data, the only thing that needs to be communicated to them is the location and value of the non-zero element.

**Example 2.1.** Example of how operations affect the user’s transformation matrix when using permutation matrix scheme.

<table>
<thead>
<tr>
<th>operation</th>
<th>( \mathbf{x}^{(s)} )</th>
<th>( \mathbf{A}^{(s)} )</th>
<th>( \mathbf{x}^{(s)} \mathbf{A}^{(s)} )</th>
</tr>
</thead>
</table>
| -               | (1, 5, 7, 2, 9)        | \[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\] | (1, 5, 7, 2, 9) |
| deletion at 3   | (1, 5, 2, 9, 0)        | \[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0
\end{pmatrix}
\] | (1, 5, 0, 2, 9) |
| deletion at 2   | (1, 2, 9, 0, 0)        | \[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{pmatrix}
\] | (1, 0, 0, 2, 9) |
| insertion of 5  | (1, 2, 5, 9, 0)        | \[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0
\end{pmatrix}
\] | (1, 5, 0, 2, 9) |

Each node can calculate its part of the difference itself, so the user needs only to communicate \( O(\log \ell + \log q) \) bits corresponding to the location of the insertion or deletion and the value affected. The user needs to be able to restore the actual data from the transformed data and for that he needs to know what his transformation matrix \( \mathbf{A}^{(s)} \) is. Fortunately he doesn’t have to store the entire matrix as he can deduce the it by knowing the history of the locations his operations affected, however doing so will still incur \( O(\log \ell) \) storage overhead per edit.

**Example 2.2.** This example shows how this augmentation would work with the linear coding scheme specified in example 1.2.
3 Overcoming storage overhead

The drawback of the permutation matrix based approach is the $O(\log \ell)$ storage overhead. When the number of edits becomes large, the storage savings given by utilizing coding will be eliminated by the stored edit history. Fortunately there are approaches that can alleviate the issue.

3.1 Hybrid scheme

First pick some value $\alpha$ such that $0 \leq \alpha \leq 1$. Next divide the data of each user into two parts of size $\alpha \ell$ and $(1-\alpha)\ell$. Then apply the permutation matrix scheme on the first part and use no input transformation on the second part. The average communication cost will be $O(\alpha(\log(\alpha\ell)+\log q)+(1-\alpha)^2\ell\log q)$ and the average storage overhead per edit will be $O(\alpha\log(\alpha\ell))$. While this scheme allows control over the trade-off between storage overhead and communication cost, it doesn’t put a cap on the total storage overhead and the communication cost becomes too large too fast.

3.2 ”Flushing” based approach

In this scheme we pick some integer $p$ and reinitialize the permutation scheme after every $p$ operations. Specifically we use the permutation matrix scheme as described until some user has performed $p$ operations. After user $s$ has performed $p$ operations, he retrieves his data from the nodes and uses his known history to construct the current state of his data. Then he sets his transformation matrix $A^{(s)}$ to be an identity matrix. Next he communicates $O(\ell\log q)$ bits to the nodes, so they could update their data to match his new input to the encoding function given by $x^{(s)}A^{(s)}$. Now the user can safely delete his operation history and continue using this scheme.
The maximum amount of extra data the user has to store will be $O(p \log \ell)$ (lower if you combine this approach with the hybrid scheme). Without this approach the overhead might grow infinitely large. The user will perform the "flush" after every $p$ operations, communicating $O(\ell \log q)$ bits in the process, so the average communication cost will be $O(\log \ell + \log q + \frac{\ell \log q}{p})$. This is larger than the $O(\log \ell + \log q)$ of the vanilla permutation matrix scheme, but it decreases as $p$ increases. This approach adds a cap to the total storage overhead that the previous schemes lacked, and additionally it allows for a configurable trade-off between communication cost and total storage overhead.

Example 3.1. Some possible picks for $p$ and the resulting costs.

<table>
<thead>
<tr>
<th>$p$</th>
<th>Avg. Communication Cost</th>
<th>Total storage overhead cap per user</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O(\ell \log q)$</td>
<td>$O(\log \ell)$</td>
</tr>
<tr>
<td>$\log \ell$</td>
<td>$O(\frac{\ell \log q}{\log \ell})$</td>
<td>$O(\log^2 \ell)$</td>
</tr>
<tr>
<td>$\frac{\ell}{\log \ell}$</td>
<td>$O(\log \ell \log q)$</td>
<td>$O(\ell)$</td>
</tr>
<tr>
<td>$\ell$</td>
<td>$O(\log \ell + \log q)$</td>
<td>$O(\ell \log \ell)$</td>
</tr>
<tr>
<td>$\ell^2$</td>
<td>$O(\log \ell + \log q)$</td>
<td>$O(\ell^2 \log \ell)$</td>
</tr>
</tbody>
</table>

4 Acknowledgements

This report is based on the paper "Synchronizing Edits in Distributed Storage Networks" written by Salim El Rouayheb, Sreechakra Goparaju, Han Mao Kiah and Olgica Milenkovic. The aforementioned paper is freely accessible from the link: [http://arxiv.org/pdf/1409.1551.pdf](http://arxiv.org/pdf/1409.1551.pdf) The purpose of this report is to give a short overview on the aforementioned paper and a reasonable understanding of the parts of the paper deemed most interesting by the author of this report. Everything expect the "Flushing" based approach is based on the aforementioned paper, the "Flushing" approach is an addition by the author of this report. This report was made possible with the help of Prof. Vitaly Skachek.